(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
20 February 2003 (20.02.2003)

PCT

(10) International Publication Number
WO 03/014917 A1

(51) International Patent Classification[7]: G06F 9/00, 9/44, 15/00

(21) International Application Number: PCT/IL02/00652

(22) International Filing Date: 8 August 2002 (08.08.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/310,868 9 August 2001 (09.08.2001) US

(71) Applicant *(for all designated States except US)*: **PRIME-LAYER SOFTWARE INC.** [US/US]; 615 South DuPont Highway, County of Kent, Dover, DE 19901 (US).

(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: **NIDDAM, Albert** [IL/IL]; 1 David Elazar St., Givat Shmuel 54032 (IL).

(74) Agent: **FRIEDMAN, Mark, M.**; Beit Samueloff, 7 Haomanim St., Tel Aviv 67897 (IL).

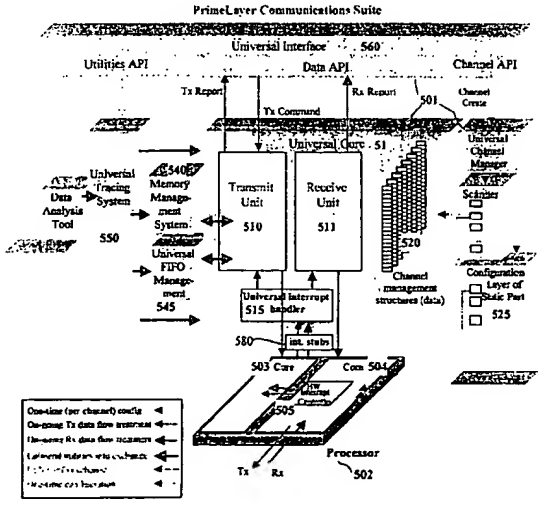
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*
— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: A UNIVERSAL DRIVER APPARATUS AND METHOD FIEL AND BACKGROUND OF THE INVENTION



(57) Abstract: A universal software driver apparatus (PCS), and method for use thereof, that enables universal data handling for a variety of protocol-processor pairs. At the core of the PCS is the premise that in various protocol driver software codes, many of the customized functions (protocol and processor specific) feature similar structures or patterns. The PCS provides a universal structure for these functions, which supports a variety of protocols and processors (502) and enables a plurality of physical channels supported by the processor (502). Additional embodiments of the PCS include adding functional components to the universal core (501), including a scanner component, initialization component, memory management system (540), universal tracing system (550), data analysis tool and/or a universal FIFO management component (545).

WO 03/014917 A1

A UNIVERSAL DRIVER APPARATUS AND METHOD

FIELD AND BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a system and method for data communications, and in particular, to a unifying method for servicing multiple data protocols and multiple processors.

2. Description of the Related Art

The demand for intelligent, converged networking has led to the development of increasingly advanced networking equipment that supports heavy-throughput, multi-protocol voice and data traffic. These new equipment designs combine embedded communication software supporting multiple protocols such as ATM, Ethernet/IP, and HDLC as well as next-generation processors. Currently, developers of such advanced communications equipment must incorporate both microprocessors (Communications Processors, Digital Signal Processors, Network Processors, CPUs) and embedded protocol software (stacks) into their hardware designs. Such designs, however, are an increasingly difficult and frustrating proposition for equipment vendors, that typically causes both mounting and unpredictable development costs and unacceptable time-to-market delays. Custom-tailored solutions are inefficient, unreliable and are not portable from design to design.

According to the seven-layer OSI model, the software closest to the hardware (located in the Data Link and Physical layers) interacts directly between the hardware and the upper protocol software stacks. This driver-layer or "foundation layer", which is typically referred to as Layers 1-1 ½ of the seven-layer OSI model, handles bi-directional communications for communications devices. The driver-layer software is primarily responsible for enabling successful transmission of information from the protocol stacks to the hardware, and vice versa.

Although the above-described OSI (Open Systems Interconnection) layer was officially adopted as an international standard by the International Organization of Standards (ISO), it should be noted that ATM, which is a standard for cell relay (from the International Telecommunications Union- Telecommunication Standardization Sector (ITU-T)) does not exactly respect the OSI seven layers model. Instead, ATM defines its own model. For ATM, the equivalent of the driver layer, or layers 1-1 ½ (when stated in this document) should be

considered as made of the following layers or sub-layers (as a non limiting example): PHYSICAL Layer, ATM Layer, SAR Sublayer, and Common Part Convergence Sublayer (CPCS).

In past years, development of layers 1-1 ½ for popular processors such as Motorola's MPC 302 and MPC 360 was relatively straightforward. It has only been recently, as companies have migrated to newer processors with more complex, on-silicon communications peripherals, such as Motorola's PowerQUICC family, that they have experienced tremendous difficulty in developing foundation layer solutions. In addition, there are several basic processor families that require servicing; for example: communication processors, DSP (Digital Signal Processors), network processors and simple processors (pure CPUs, for example, the Pentium of Intel) etc. There are similarly a variety of communication protocols that require processing by the above mentioned processors, including Ethernet, HDLC, ATM, UART, Transparent, BISYNC, AppleTalk and SPI etc.

Figure 1 illustrates various typical software functionalities of this driver-layer or foundation layer software. As can be seen in the figure, a driver **11** receives information units such as cells, frames, and packets from the protocol stacks **12** (these stacks use the services of a real-time operating system (RTOS) **13**) and places them on the appropriate hardware channels (the physical channels that connect peripheral devices to processors). The driver **11** also receives incoming information from the hardware channels and passes it on to the protocol stacks **12** for treatment. The driver-layer is responsible for communication register setting, initial protocol-to-channel integration, dynamic handling of buffers and buffer describing structures, interrupt handling, and queue management. The board support package (BSP) **14** is used only through its interrupt stubs that perform the first basic treatments associated with any interrupt. These interrupt stubs subsequently invoke the interrupt handling function of the drivers.

Like communication between the upper layers (Layers 2-7 of the OSI model), the driver layer's interaction with the upper protocol stacks is standards-based, specific software-to-software communication, which is appropriate for every driver-protocol pair. For example, an Ethernet driver requires particular software to process data from/to an Ethernet channel, and similarly for each protocol stack. In addition, the driver layer must also contend with an infinite variety of hardware configurations below, and therefore every processor requires specific driver

software when processing data from/to a protocol stack. Complicating matters further is the fact that each protocol, such as IP, ATM and HLDC etc., requires different mechanisms of physical-channel interaction, as different types of information units (packets, frames, cells, etc.) travel in different sequences and formats. As a result of the above complications, drivers are typically designed separately for each protocol stack, and are tailored to the chosen processor as well as the number of physical channels (PHY). Every pair of protocol-processor therefore typically has a specialized driver, and any computing device that requires processing ability for numerous protocol-processor pairs requires a similar number of specific drivers.

Typical implementations using the driver layer have, as described above, become increasingly complex and have become a leading cause of a number of mission-critical problems for communications equipment vendors. The current industry standard, as can be seen with reference to **Figure 2,** is based on drivers 22 that are designed separately for each protocol stack 24, and are tailored to the chosen processor 26. An additional constraining factor may be the number of physical channels (links) – some drivers are so inflexible that simply adding another channel (of the same protocol) to an equipment design means duplicating driver code so that identical drivers run the two channels. Because these software designs are so hardware-dependant, there currently are very few alternatives for an equipment vendor other than to either: develop them from "scratch"; or if and when available, buy non flexible, non efficient and non reliable drivers; or

modify drivers from prior projects; and then test and debug the drivers in-house.

The industry-standard process of custom-designing, developing, integrating (horizontally and vertically), testing, debugging, benchmarking and improving multiple protocol drivers at the foundation layer is not efficient and often leads to the following critical problems:

i. Performance: Multiple protocol drivers compete with each other for use of the processor's instruction cache. When one stack's driver is operating, and the processor receives an interrupt for transmission/reception via another protocol, the processor will be in a state of "cache miss". This results in critical real-time performance lapses that occur as the processor first unloads the current driver and then loads the appropriate functions for the interrupting protocol into its cache. Today's communication processors make very poor use of their cache.

ii. Development cost: Equipment vendors must employ teams of expert embedded software developers and deploy them time and again to develop, integrate and debug customized driver-

layer solutions for each new product design. The equipment vendor incurs additional expenditures if s/he licenses driver source code from third parties, if and when available, which must then be further modified, integrated and tested.

iii. Reliability: Custom-designed protocol driver solutions have by definition never been field-tested and do not ensure field reliability.

iv. Time in market: Because driver-layer software is currently tailor-made to a processor, protocols and number of channels, a proposed modification of any of these variables in the equipment design will require repetition of the lengthy software design, development, modification, integration, debug measurement and optimization cycle.

v. Time-to-market delays: Development, debugging, horizontal integration and testing of such tailor-made software are unpredictable and can typically take up to 18 months, depending on device configuration and design complexity.

vi. Development risk: Because the customized driver layer has not been tested or debugged, the software team will be required to respond to equipment and software failures as they occur during testing and then attempt to determine whether the source of the problem is in the drivers, and if so, to identify the exact nature of the problem and debug the software accordingly.

There is thus a widely recognized need for, and it would be highly advantageous to have, a universal system and/or method that can enable reception and transmission of data units from different protocols simultaneously. In addition, it would be highly advantageous for such a system or method to interoperate with various processors in a unified way, so as to enable an efficient, reliable and portable solution for the above-described problems associated with driver software.

SUMMARY OF THE INVENTION

According to the present invention there is provided a revolutionary foundation layer software solution, hereinafter referred to as the "PrimeLayer Communication Suite" or PCS. The PCS is a single, unified "black box" protocol driver software solution that interoperates with numerous processors (such as Communications Processors, Digital Signal Processors, Network Processors and/or CPUs), multiple protocols, any number of channels (links), real-time operating systems (RTOS) and Board Support Packages (BSPs). In addition, the PCS features fast, run-time configuration that enables rapid setup. This configuration does not require integrators of the

present invention to modify the software's source code, thus eliminating human-error configuration bugs. The streamlined, unified architecture also increases real-time performance, reliability and is portable from design to design. In the following description, the terms "channel" and "link" are used in an interchangeable way.

At the core of the PCS is the premise that in protocol driver software, many of the customized functions (protocol and processor specific) feature similar structures or patterns. The present invention provides a universal structure for these functions that supports all protocols and processors and enables any number of physical channels supported by the processor. All protocol- and processor-specific requirements are already pre-integrated with the product's universal core. As a stand-alone, product, the PCS design allows it to be portable from one project design to another without modification of the PCS source code, providing revolutionary predictability and stability.

According to the present invention there is provided a unifying method that, when implemented as software, enables the servicing of several different protocols (at the same time) and different kinds of processors. The service offered by this software, together with the chosen processor, corresponds to the functionalities of Layer 1-1 ½. That is, it corresponds to the ongoing dynamic handling of the data units for both reception and transmission. The setting up of the universal driver, according to the present invention, is as follows:

i. distinguishing between dynamic and static layers for a plurality of drivers;

ii. transferring of dynamic data processing means from the plurality of driver programs to fields of a Channel Management Structure;

iii. unifying the dynamic functionalities from the plurality of drivers into a single unified driver; and

iv. maintaining the static layers from the plurality of drivers in a universal driver.

The resulting universal driver is typically smaller in size (code footprint) than a single standalone driver, allowing the placing of such a universal driver in the instruction cache of a processor, thereby enabling dramatically enhanced processing ability, and allowing significant reductions in resource requirements.

According to an additional embodiment of the present invention, in the case where processors are used that contain no hardware com part, such as pure or simple CPUs, the present

invention performs the com functions through any software controller and/or hardware component that performs the com functionalities.

According to a preferred embodiment of the present invention, a method is provided for enabling data transmission between a plurality of protocols and processors. The method includes:

i. setting up a global management structure, for storing specific protocol-processor pair variables upon initializing of a specific protocol-processor channel, *by a global context configure function;*

ii. configuring a static part of at least one such specific channel, by a configuration layer of static part module, and writing the static part to memory (internal or external processor memory);

iii. scanning the memory to read at least a part of the static part of the channel, by a scanner component, and using this part to create a data management structure for supporting the specific channel, by the scanner component;

iv. filling the data management structure with variable parameters for the specific channel, by the scanner component;

v. transferring a data command to the kernel of the global management structure, from the data management structure; and

vi. executing data handling for the specific protocol-processor pair, by the global management structure kernel.

Further embodiments of the present invention include adding functional components to the universal core, including a scanner component, initialization component, memory management system, universal tracing system, data analysis tool and/or a universal FIFO management component.

According to an additional embodiment of the present invention, a universal Application Program Interface (API) is provided, for enabling a universal interface for ongoing operation with any protocol/processor, using the same interface.

BRIEF DESCRIPTION OF THE DRAWINGS

The principles and operation of a system and a method according to the present invention may be better understood with reference to the drawings, and the following description, it being understood that these drawings are given for illustrative purposes only and are not meant to be limiting, wherein:

FIGURE 1 is an illustration of various typical software functionalities of the driver-layer software.

FIGURE 2 is an illustration of a typical driver layer implementation.

FIGURE 3 is an illustration of the typical driver functionalities, and the unifying process of these functionalities, according to the present invention.

FIGURE 4 is an illustration of the PCS driver layer implementation, according to the present invention.

FIGURE 5 is an illustration of a preferred embodiment of the PCS, according to the present invention.

FIGURE 6 is an illustration of the Channel management Structures of the PCS.

FIGURE 7 is an illustration of a typical implementation of DSL technology in a data network.

FIGURE 8 is an illustration of an implementation of DSL technology in a data network with a universal driver, according to the present invention.

FIGURE 9 is an illustration of the global architecture overview, reflecting the incorporation of the universal scanner, according to the present invention.

FIGURE 10 is an illustration of the method whereby a communications processor functions in a multi-protocol environment, according to the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The present invention relates to a system and method for providing a single, unified "black box" protocol driver software solution that interoperates with numerous common processors (Communications Processors, Digital Signal Processors, Network Processors and/or CPUs), multiple protocols, any number of physical channels (links), real-time operating systems (RTOS) and Board Support Packages (BSPs).

The following description is presented to enable one of ordinary skill in the art to make and use the invention as provided in the context of a particular application and its requirements. Various modifications to the preferred embodiment will be apparent to those with skill in the art, and the general principles defined herein may be applied to other embodiments. Therefore, the present invention is not intended to be limited to the particular embodiments shown and described, but is to be accorded the widest scope consistent with the principles and novel features

herein disclosed.

Specifically, the present invention provides a streamlined, unified architecture that increases real-time performance, reliability and is portable from design to design. At the core of the present invention is the premise that, in protocol driver software, many of the customized functions (protocol and processor specific) feature similar structures or patterns. The present invention provides a universal structure for these functions that supports all protocols and processors and enable any number of physical channels supported by the processor. All protocol and processor-specific requirements are pre-integrated with the product's universal core. As a stand-alone, product, the PCS design allows portability from one project design to another, in run-time without modification of the PCS source code, providing revolutionary predictability and stability.

In the present document, the following conventions (concerning the naming policy) have been adopted:

i. The examples of function names are written in an *Italic* style, with each word beginning being UpperCased (e.g. *ExampleOfFunction).*

ii. The parameters are typically lightly shaded, for example, *ExampleOfFunction* (parameter)

iii. The variables, when needed, include "underscore" separations (for example, example_of_variable)

The principles and operation of a system and a method according to the present invention may be better understood with reference to the drawings and the accompanying description, it being understood that these drawings are given for illustrative purposes only and are not meant to be limiting, wherein:

Figure 3 illustrates in **stage 1** a classical configuration for a multi-protocol application, wherein one driver is used per protocol supported. As can be seen in **stage 2** of the figure, when the classical drivers **31** are carefully examined, each driver has 2 Sub-Layers that can be distinguished:

i- an upper layer **32** corresponding to the ongoing dynamic handling of the data units (ongoing data flow management); and

ii- a lower Layer **33** corresponding to the static initialization of the channels (code that runs once, at the very initialization of a channel).

Further examining the upper layer 32 previously defined, it appears, as can be seen in stage 3, that the functionalities 34 serviced by all the different protocol drivers are the same. These include: Buffer Handling; Frame Handling; Buffer Descriptors Handling; Interrupt Handling (Rx, Tx, Errors); Transmit (Tx) Commands; Receive (Rx) Commands; Tx Reports; Rx Reports and Error Reports etc. Furthermore, the data units handled by these functionalities 34 are buffers, and for all protocols, a buffer can be seen as a collection of bytes with a start address and a length.

Another important principle used in the unifying effort is that the differences (from protocol to protocol) between the treatments/means 35 (performed by the lower layer 33 to support the functionalities 34 listed above) can be moved from the program (driver software) itself to a data plane (a database) 37 storing the specific parameters (variables) to be used by the universal driver. This database, hereinafter referred to as the "Channel Management Database" (CMD), comprises the various Channel Management Structures (one structure is provided per channel), which store the specific parameters (treatments) required by each individual channel. In this way, the individual driver data treatment means are stored in a unified database to be used by a universal driver, and not in driver software code for individual drivers.

Remaining focused on these unifying principles, it is then possible, as can be seen in stage 4 of the figure, to unify all the upper layers 32 of the various drivers, according to their common functionalities. The result is a universal driver 36 that is the core of the present technology, and the primary added value. It should be noted that the actual size of this universal driver 36 (unified upper layer) is much smaller than the typical size of the upper layer of a single driver 32. This is achieved due to the small size of the unified dynamic handling code (code footprint) 37, which is so small that it can be fitted into the instruction cache of the processor. This unified code, moreover, does not need to be rewritten each time for a new processor or a new protocol.

As can be seen in stage 5, the lower layer 33 includes code 38 that is highly protocol and processor dependent. This code 38 must therefore be able to know each initialization bit, in order to select the processor pins used and the baud rate used, etc. At this level, only limited unifying work can be done (mainly per family of processors/protocols). This protocol and processor dependent code is not critical from the point of view of real time performance, because it runs only once, at the initialization of each channel (link).

The above described method for setting up the universal driver, according to the present invention, can be summarized as follows:

i. Distinguishing between dynamic (on going treatments during the whole life of the channel) and static (executed only once at the creation of the channel) layers for a plurality of drivers.

ii. Transferring of dynamic data processing means from the plurality of driver programs to fields of a database containing a Channel Management Structure for each channel.

iii. Unifying the dynamic functionalities from the individual drivers into a single universal driver (unified upper layer).

iv. Maintaining the static layers from the plurality of drivers in the universal driver.

Figure 4 illustrates a preferred embodiment of the PCS, according to the present invention. As can be seen in the figure, the various protocol drivers have been replaced by the PCS universal driver layer **41**. This universal driver layer **41** thereafter processes all data requests and transfers, for all types of protocols **42**, processors **43**, RTOSs **44** and BSPs **45**.

There are several principles for implementing the PCS:

i. The PCS is event driven, and the main events driving the PCS are: Channel Create; Transmit Command and Hardware events.

ii. The two events *Channel Create* and *Transmit Command* occur whether at task or interrupt level. Their relative priorities depend on the specific application architecture.

iii. The *Hardware events* (the most common events) pace the overall system behavior through interruptions.

iv. The default rule is: The interruptions have an absolute priority over any task level code.

v. The exception is any particular section of a task that (for any reason) masks the interrupts (non interruptible section of a semaphore for example).

vi. Because of the previous rules, the PCS can be viewed as hardware and software (interrupts/tasks) driven.

Figure 5 illustrates the actual components of the PCS (universal core) **51** as well as further optional components. The PCS **501** provides all necessary driver-layer functions required to transmit information units received from the protocol stacks and to receive and forward incoming messages to the relevant protocol stacks.

1. Processor 502 (not part of the PCS):

The processor **502** continuously interacts with the PCS **501**. Any communication-optimized processor (for example, the: MPC302, MPC 360, MPC860, MPC8260 of Motorola, MSC8101 DSP of Motorola/Agere, 405GP and NPe405 of IBM) includes a core component **503**, for executing the program, performing basic arithmetical operations; and a "com" **504** component that performs basic operations related to the communication requirements of the processor, corresponding to the protocol controller level. The com component typically includes a hardware block (interrupt controller) **505** that handles the different relative priorities of the hardware events. Examples of com part functionalities include managing a protocol state machine, computing a CRC, encapsulating a message within a protocol frame (adding header and tail), sending a message (to Hardware FIFO), receiving a message (from the Hardware FIFO) and generating interrupts, etc. The particular behavior of the interrupt controller **505** depends on the processor architecture, but in any case, it presents to the PCS **501** only the interrupt of the highest level.

In any dialog between a core part and a com part of a processor, the semaphores play an important role: The **semaphores** are used anytime two entities compete for the same resource. Semaphores are widely used and well known in the art, and are used in most applications. A semaphore is implemented through a bit (that has an address) whose value 1 has a meaning (for example, "semaphore not available"), and whose value 0 has the other meaning (for example, "semaphore available"). From a general point of view, to set a semaphore may simply mean to set a certain bit (at a certain address) to a certain value. However, since the exact location of a semaphore, and its exact values (for both unavailable and available semaphores) depend on the protocol/processor, these values, according to the present invention, can be set once, at the initialization of a channel (and within the channel management structure). Since the semaphore code does not require specific location values etc., according to the present invention, it is considered universal. Usage of such universal semaphore code enables more efficient usage of processing resources. For example, semaphores are widely used to organize the work performed on BDS by both the core and the com. According to the present invention, any semaphore used is identified by its address, the number of the bit used (through a mask), and the values meaning "not available" and "available". All these values are stored in the channel management structure.

2. Universal Core (PCS) 501:

The Universal Core 501 manages all changing pointers in the processor's internal and external memory, handling all dynamic transmit/receive functionality in one unified engine. The Universal Core 501 is responsible for all dynamic handling of buffers and buffer data structures and interrupt handling, and performs foundation layer error-checking of messages. The PCS includes the Transmit unit 510, the Receive unit 511, the Universal Interrupt Handler 515 and the Channel management Database 520.

3. The Universal Interrupt Handler 515:

This PCS software component knows how to handle any interrupt, whether simple or complex, generated by any processor. A complex interrupt can be, for example, an interrupt that reports about a number of sub-events, organized for example in one queue or in several queues. In such a case, using a bit rolling scanning, all the bits (events) may be rapidly checked. Such a bit rolling scanning is known in the art.

4. The Transmit Unit 510 and Receive Unit 511:

These components deal with the Transmit/Receive related basic treatments. These units optionally interface between the user concepts world (of a transmit command/ Receive report) and the ultra-general concepts of the Universal Data Receive and Transmit Manager (see below).

From the point of view of any driver, no matter what protocol or processor is being used, there is a fundamental difference between the receive and transmit processes:

The **transmit process** is initiated by the user (through a function call).

The **receive process** is (generally) not initiated by the user and typically occurs in reaction to a receive interrupt. The transmit process is analogous to initiating a phone call, and the receive process to receiving a call, which occurs at unknown times.

Any driver, no matter which protocol it supports or which processor is used, deals with data units. These data units are stored in data buffers, which are almost never handled directly. Instead there are always some helping structures, which are hereinafter referred to as "**Buffer Data Structures (BDS)**" that are used to handle the data buffers. The Buffer Data Structures (BDS) of a given channel are assumed to be accessed through pointers stored in the channel management structure, for a given channel. In any driver, no matter which protocol or processor is being used, the BDS typically comprises at least the following fields: The address of the corresponding data buffer (or data unit); the length of the corresponding data buffer; a semaphore; and the address of the next BDS of this channel (when there is no other way to

determine this address automatically. In some processors, for example, the BDS are contiguous, and in such a case there is no need of a pointer to the next BDS).

From the point of view of the core, the transmit process requires placing the data unit address in a BDS and setting its semaphore to instruct the com to actually perform the transmission. From the point of view of the com, the transmit process requires polling the semaphore, such that if it the semaphore is set, the com transmits the data units whose addresses are in the BDS. This transmission, done by the com, deals mainly with the electric levels of the corresponding protocol (but may also include some segmentation, CRC calculation and/or encapsulation/ etc.). When the transmission is completed, the com clears the semaphore and generates a transmit interrupt for this channel.

From the point of view of the com, a receive process requires choosing the first BDS with a semaphore set as available, and translating the received electric signal into a received data unit that is copied at the data buffer address provided by the BDS. It then clears the semaphore to indicate that a data unit has been received, and generates a received interrupt for this channel. From the point of view of the core, the receive process requires determining which BDS needs servicing. The core reports to the upper layer that a data unit has been received on this channel (and provides the address provided by the BDS). The core then gets the address of another free buffer and places this address in the BDS. The core then sets the BDS' semaphore to indicate that this BDS can be used again by the com.

In any driver, no matter what protocol or processor is being used, there is an Application Program Interface (API) to generate a transmission (there may also be an API to poll the reception). However, there is an "interrupt" function that is called automatically when a data unit has been received. A similar "interrupt" function is called automatically when a data unit has been transmitted.

5. The Universal Data Receive and Transmit Manager (optional- not in figure):

Significant functionality from both the Transmit unit **510** and the Receive unit **511** can be combined and effectively managed by a universal Data Receive and Transmit Manager. For example, the way the next BDS is determined, the way a semaphore is released, and the way some information is updated within a BDS, etc. For a given channel, the Universal Data Receive and Transmit Manager behaves according to the general behavior parameters of this channel (provided by the Scanner **530**). This component only assumes that some data units (buffers) have

- The Universal Interrupt handler 515 analyses the event word (using masks read from the channel management structure 520) and determines whether the interrupt currently serviced was a receive, a transmit or an error.
- In the case of transmit interrupt:
    1. Channel_id points to a channel management structure. One of this structure's fields is the pointer to the buffer data structure (BDS) to be handled during the next transmit interrupt (named for example ptr_to_tx_bds_first_to_handle).
    2. Variable functions, read from the channel management structure, are used to get this BDS information:
        - ptr_to_transmitted_data
        - length.
    3. Another field of the channel management structure is a user defined variable function (named *TxReport*).
    4. This function is invoked with the previously extracted BDS information.
    5. Steps 2 to 4 are repeated as long as there is a Transmit BDS that must be handled.
- In the case of receive interrupt:
    1. Channel_id points to a channel management structure. One of this structure's fields is the pointer to the buffer data structure (BDS) to be handled during the next receive interrupt (named for example ptr_to_rx_bds_first_to_handle).
    2. Variable functions, read from the channel management structure, are used to get this BDS information:
        - ptr_to_received_data
        - length.
    3. The previous information is saved in temporary variables.
    4. Another field of the channel management structure is a user defined variable function used to allocate buffers (named GetBuffer)
    5. This function is invoked to get a pointer to a free Buffer.
    6. This pointer is saved in the BDS.
    7. The semaphore associated to this BDS is set to enable its use by the com, for another reception.

8. Another field of the channel management structure is a user defined variable function (named RxReport).
9. This function is invoked with the previously extracted BDS information that was saved in the temporary variables.
10. Steps 2 to 9 are repeated as long as there is a Receiced BDS that must be handled.

- In the case of error interrupt:
    1. A field of the channel management structure is a user defined variable function (named ErrorReport).
        - This function is invoked with the event word received from the interrupt stub.

The PCS, heretofore described, enables the following features:

i. Processor-independent: Supports any Communication Processor, Digital Signal Processor, Network Processor or simple processor (CPU).

ii. Protocol–independent: Supports all protocols, and any number of channels, that are supported by the processor.

iii. RTOS–independent: Supports all RTOSs, and may also be run as "stand alone", without any RTOS.

iv. Run-time configuration: All integration of the PCS occurs at run-time (Plug & Play). No modification of the PCS source code is required.

v. Portable: The same code supports new hardware configurations and product designs.

The PCS can be delivered according to the following ways:

i. The PCS is delivered to the customer as object code on CD-ROM, or can be downloaded from a data network (such as the Internet, Intranet, Extranet etc.).

ii. A PCS integration API is provided as a .h file to direct the run-time integration of the PCS. The API contains parameter declarations for stack-level function calls.

iii. The PCS's unified functionality enables easy integration with protocol stacks, requiring minimal developer interaction. Integration features include:

a. API only: The entire interface between protocol stacks and the PCS is via API functions.

b. Object code: No intervention with PCS source code is required.

c. Minimal developer interaction: Minimal adjustments are necessary to stack source code to call the PCSs API functions.

The PCS also contains pre-programmed universal interfaces that enable automatic interoperation with any RTOS, BMS, and FIFO system and any PHY compatible with the processor.

Figure 6 further illustrates the Channel management Database **62** of the PCS. This database contains the various Channel management Structures **61** that are exactly the same variables mentioned in the above description of figure 3, where the differences (from protocol to protocol) between the treatments performed to support the functionalities listed above, can be moved from the driver program itself to the variables used by the program. Each of the Channel Management Structures **62** in the Channel Management Database, as shown in the figure, stores data that relates to the various communication protocol-processor pairs requiring processing. These are the only component of the PCS that are directly protocol-processor relevant.

Figure 6 further illustrates an additional embodiment of the present invention, wherein additional (optional) elements are operationally connected to the PCS, including: a scanner **66** and an initialization manager **68**.

Figures 7 and 8 illustrate examples of implementation of the PCS technology, for performance Enhancement in a DSL modem. Achieving high performance at low cost is crucial for manufacturers of DSL modems. **Figure 7** illustrates a typical implementation of a DSL modem, using multiple drivers. As can be seen in the figure, the architecture typical arch (separate foundation software) involves an intense struggle on the hardware resources. In contrast to this, **Figure 8** illustrates the unique CacheIN™ technology of the PCS, which enables the location of the dynamic functions of the universal driver within the instruction cache, thus avoiding "caches misses" as described above. This implementation dramatically increases the performance of a DSL modem, by driving all of the device's protocols from the processor's instruction cache, thus eliminating extra-delays due to constant "cache misses" common in current separate-driver solutions. Embedding the PCS software frees substantial CPU power from the processor, and the unique architecture of the PCS solution solves the conflicts on the hardware resources (instruction cache and address and data buses). In-house benchmark testing showed up to a 250% throughput improvement in a typical configuration (for a VDSL modem), with a Motorola MPC862P processor. This extra power can be used to provide either enhanced

services (stacks' upper layers) and/or sustain greater data rates on the modem's lines. In configurations without the PCS, the architecture (separate foundation software) involves an intense struggle on the hardware resources.

Figure 9 illustrates the Global architecture overview of the network apparatus according to the present invention. In particular, this figure illustrates the primary Scanner function, which is, for a given channel, reading the processor for driver specific settings, and placing these settings in the relevant channel management structure. As can be seen, according to the scanner-based embodiment of the present invention, Layer 1-1 ½ requires organization into three main blocks: the kernel **91**; the scanner **92**; and the static initializations **93**.

The following should be noted:

i. The kernel is the only part of the apparatus whose code is completely independent of the processor/protocol used (besides the optional utilities like the Universal FIFO management, the Memory Management System and the tracing system).

ii. The kernel is accessed on an ongoing basis (hence it is also referenced to in the present document as the dynamic part).

iii. The scanner is able to handle data from a multitude of processor types.

iv. The scanner reads part of these settings and writes them into the structures of the kernel.

v. The static initializations may be integrated into the core apparatus of the present invention, or maintained as external components/processors.

vi. The static initializations and the scanner are invoked only once per created channel (hence, it is also referenced in this document as the static part). The static initializations write some settings within the processor registers (and also sometimes in the external memory).

The method for enabling data transmission between a plurality of protocols and processors operation, according to the Global architecture of the present invention, is as follows:

i. setting up a global management structure, for storing specific protocol-processor pair variables upon initializing of a specific protocol-processor channel, *by a global context configure function;*

ii. configuring a static part of at least one such specific channel, by a configuration layer of static part module, and writing the static part to memory (internal or external processor memory);

iii. scanning the memory to read at least a part of the static part of the channel, by a scanner component, and using this part to create a data management structure for supporting the specific channel, by the scanner component;

iv. filling the data management structure with variable parameters for the specific channel, by the scanner component;

v. transferring a data command to the kernel of the global management structure, from the data management structure; and

vi. executing data handling for the specific protocol-processor pair, by the global management structure kernel.

The reason that the scanner reads something that has been written in a previous phase, causing an apparent inefficiency, is because in typical drivers, there is no strong separation between the static initializations, the scanner and the kernel itself. Everything is mixed together. If all the elements are mixed, it is impossible to isolate a kernel that is independent of the processor/protocol (the case of all today's drivers) because the static initializations are very highly processor/protocol dependent. On the other hand, if all the elements are separated, as required according to the present invention, the kernel can be organized in a way that its code is independent of the processor/protocol. Furthermore, the static initializations can then be inherited from any third party source (sometimes the chip makers themselves provide such free code).

IMPLEMENTATION OF THE PCS

i. Important data structure: global_management_structure

The global_management_structure is a global structure that stores the variables that are set once at the very initialization of the system, by the function *GlobalContextConfigure*. This structure should comprise:

- processor_name
- processor_internal_registers_address
- *UserProvidedInitializationAlloc*

  to get the memory amounts needed by the PCS for its own internal needs

ii. Important data structure: channel management structure (Data Management Structure)

In order to support a specified channel (link), the software first invokes scanning functions (comprised in the software implementing the present invention) that read the static initializations from the processor and store some information needed by the software in a portion of memory dedicated to this specified channel. This portion of memory is hereinafter referred to as the "channel management structure". Each channel has an associated channel management structure. The channel management structure also includes many behavior fields that describe how the channel has to be managed. The kernel of the software implementing the present invention regularly accesses these fields in order to dynamically determine which handling is needed by this channel.

The fields of the channel management structure comprise a description of:

- Some identifying information for this channel:
  - The PHY type of this channel (channel_phy_type)
  - The number of this channel (channel_number)
  - The protocol supported by the channel
- Some fields enabling the dynamic handling of the data buffers used by the receive and the transmit parts of this channel.

  For example:
  - pointer to the first buffer data structure (BDS) used by the receive
  - pointer to the buffer data structure (BDS) currently used by the receive
  - pointer to the first buffer data structure (BDS) used by the transmit
  - pointer to the buffer data structure (BDS) currently used by the transmit
  - function variable (also initialized during the creation of the channel) that enables the stepping to the next BDS (maybe a quite obvious function when the BDS are contiguous)
- Some addresses and masks enabling the handling of the semaphores used for this channel by the core part and the com part, for example:
  - pointer to the exact position of the semaphore (within the BDS)
  - mask of the bit (or bits) used as a semaphore within the address pointed by the previous field

In an implementation of the present invention, it may be recommended to use C programming language, instead of C++, in order to spare the encapsulating overhead involved by C++. Furthermore, it may help understanding the method of the present invention, to view the channel (link) as an object, whose fields members are the fields of the channel management structure and whose identifier or handle is the address of the channel management structure.

iii. Static initializations

The software implementing the present invention assumes that the static initializations of the communication channels have been performed previously by any means.

The distinction between static and dynamic is the following:

- Static initializations are the ones needed to set the PHY of the channel, the pins configurations, the baud rate, the working modes, the electrical options, etc.
- Dynamic handling is the handling of the data units during the whole ongoing reception and transmission process.

There are generally many third party codes available to perform the static initializations. According to an additional embodiment of the present invention, the software also includes some modules that enable the performance of the static initializations for any channel. This part of the software is clearly processor and protocol dependent.

The Kernel: Overview

The kernel can be viewed as the most important part of the apparatus for and method of executing data communications. The kernel is the part of the Layer 1-1$^{1/2}$ responsible of the ongoing dynamic management of the data units. Any Layer 1-1$^{1/2}$ driver (whatever the protocol is and for any processor) must include at least 3 functions:

- A function that performs the transmission of a data unit. This function is invoked by the user, when he or she wants to transmit.
- A function that is invoked (at interrupt level) when a data unit has been received.
- A function that is invoked (at interrupt level) when a data unit has been transmitted.

These three functions (further described in the next paragraphs) represent the most important part of any Layer 1-1$^{1/2}$ driver. All the other functions encountered in any driver are of less importance and are generally mere tools.

Amongst the tools functions, the two "most important" are the following:

- A function to configure the global context, which is invoked only once; and
- A function to create a channel (link), which is invoked only once for each channel. These two secondary functions of the kernel (according to the invention) will also be detailed in the next paragraphs.

According to the method of the present invention, the kernel can be implemented as a set of general functions. These general functions include the following:

For any active channel [and each time that for this channel a transmit has to be done (calling interface) or each time a receive has to be handled (generally following an interrupt)] any of these functions:

- Accesses the channel management structure of this channel
- Uses the fields of the channel management structure to know how to behave
- Actually behaves accordingly

The functions of the kernel of the invention are listed below.

Table 1:

API	Number of times this function is invoked	Importance of the function
TransmitFunction	Invoked an unlimited number of times in a channel life. It means, each time a data unit: • is to be transmitted or • has been transmitted or • has been received	Main
ReceiveHandler		
TransmitHandler		
GlobalContextConfigure	Invoked only once	Secondary
channel*Create*	Invoked only once for a given channel	

In the present method, the kernel can be implemented as a set of general functions that are unified and whose code does not depend on the processor or the protocol. The other parts [the scanner and the static initializations needed to run a Layer 1-1$^{1/2}$ do depend on the protocol and on the processor]. The present method thereby enables the ongoing dynamic handling of the data units to be isolated in the kernel, organized in a way that the handling does not depend on the

particular processor used or on the particular protocol (or protocols) of the supported channel (or channels).

Kernel: Transmit function

Prototype: *TransmitFunction*(channel_id, ptr_to_data_to_be_transmitted, length)

Table 2 : General description of *TransmitFunction*

Description	Invoking
Indicates to the com part to perform the transmission of a specified data unit on a specified channel. The channel is identified by a channel_id. The data is identified by ptr_to_data_to_be_transmitted	By the user, each time s/he wants to perform a transmit.

Table 3 : Parameters of *TransmitFunction*

Parameter	Explanation
channel_id	Identifier of the channel on which the user wants to perform the transmission. channel_id is typically the pointer to the channel_management_structure of the channel.
ptr_to_data_to_be_transmitted	Pointer to the data to be transmitted. Generally this pointer is a pointer to a buffer got through an Alloc function of any Buffer Management System.
length	number of bytes to be transmitted.

Table 4: Actions performed by of *TransmitFunction*

Actions	Remark
channel_id is used to access the channel_management_structure of this channel.	channel_id is typically the pointer to the channel_management_structure of the channel.
Within channel_management_structure, this function reads the variables needed to handle the transmit.	Main variables needed to handle the transmit: • the address of the buffer data structure that the com part of the processor will used to perform the next transmit. • the addresses and the values used as a semaphoring mechanism between the core and the com part of the processor. This semaphore is used by the core part to say to the com part: "the data is ready and at the good address, you can transmit it now".
Within the buffer data structure (whose address has been read in the previous step from the channel_management_structure), this function writes at the good location the ptr_to_data_to_be_transmitted	
At the address of the semaphoring mechanism, this function writes the value needed to say to the com part to transmit.	

Kernel: ReceiveHandler

Prototype: *ReceiveHandler* (channel_id)

Table 5 : General description of *ReceiveHandler*

Description	Invoking
Reports to the upper layer that a data unit has been received on a certain channel. The received data unit is passed to the upper layer using the pointer to the received data (or any agreed identifier, for example a buffer_id). The channel is passed to the upper layer using the channel_id. Frees some resources to enable the next receptions.	Invoked following an interrupt generated each time a data unit is received.

Table 6 : Parameters of *ReceiveHandler*

Parameter	Explanation
channel_id	Identifier of the channel on which the receive interrupt has been received. . channel_id is typically the pointer to the channel_management_structure of the channel.

Table 7: Actions to be performed by *ReceiveHandler*

Actions	Remark
channel_id is used to access the channel_management_structure of this channel.	channel_id is typically the pointer to the channel_management_structure of the channel.

Within channel_management_structure, this function reads the variables needed to handle the received data.	Main variables needed to handle the receive data: • the address of the Buffer data structure where the reception has been done by the com part. • the addresses and the values used as a semaphoring mechanism between the core and the com part of the processor. This semaphore is used by the com part to say to the core part: "the data reception of a data unit has been is completed you can use the data (through a pointer)".
Within the buffer data structure (whose address has been read in the previous step from the channel_management_structure), this function reads at the good location the ptr_to_data_to_received_data	
This function then reports to the upper layer that on the channel_id , a data unit has been received, and the address of this data unit is: ptr_to_data_to_received_data	This is done by invoking a call back (user provided or user defined) function whose prototype is, for example: *ReportReceptionToUpperLayer*(channel, ptr_to_data) In such a case, *ReceiveHandler* simply invokes: *ReportReceptionToUpperLayer*(channel_id, ptr_to_data_to_received_data)
This function then invokes a user provided *GetBuffer* function in order to get the address of a new free buffer.	
This address got by the previous call (to *GetBuffer*) is inserted within the buffer data structure	
At the address of the semaphoring mechanism (got during the second phase of this function), this function writes the value needed to say to the com part that the buffer data structure is once again available to the com in order to perform a receive.	

Kernel: Transmit handler

Prototype: *TransmitHandler* (channel_id)

Table 8 : General description of *TransmitHandler*

Description	Invoking
Reports to the upper layer that a data unit has been transmitted on a certain channel. The transmitted data unit is identified using the pointer to the transmitted data (or any agreed identifier, for example a buffer_id). The channel is passed to the upper layer using a channel_id	Invoked following an interrupt generated each time a data unit is transmitted.

Table 9 : Parameters of *TransmitHandler*

Parameter	Explanation
channel_id	Identifier of the channel on which the channel has been received the transmit interrupt. channel_id is typically the pointer to the channel_management_structure of the channel.

Table 10 : Actions to be performed by *TransmitHandler*

Actions	Remark
channel_id is used to access the channel_management_structure of this channel.	channel_id is typically the pointer to the channel_management_structure of the channel.

Within channel_management_structure, this function reads the variables needed to handle the transmitted data.	Main variables needed to handle the transmitted data: • the address of the Buffer data structure from where the transmission has been done by the com part. • the addresses and the values used as a semaphoring mechanism between the core and the com part of the processor. This semaphore is used by the com part to say to the core part: "the data transmission of a data unit has been is completed. You can use the buffer data structure".
Within the buffer data structure (whose address has been read in the previous step from the channel_management_structure), this function reads at the good location the ptr_to_data_to_transmitted_data	
This function then reports to the upper layer that on the channel_id , a data unit has been transmitted, and the address of this data unit is: ptr_to_data_to_transmitted_data	This is done by invoking a call back (user provided or user defined) function whose prototype is, for example: *ReportTransmissionToUpperLayer*(channel, ptr_to_data) In such a case, *TransmitHandler* simply invokes: *ReportTransmissionToUpperLayer*(channel_id, ptr_to_data_to_transmitted_data)
At the address of the semaphoring mechanism (got during the second phase of this function), this function writes the value needed to say to the com part that the buffer data structure is once again available to the core in order to perform another transmit.	

Kernel: GlobalContextConfigure

Prototype: *GlobalContextConfigure*

(

processor_name,

processor_internal_registers_address,

UserProvidedInitializationAlloc)

Table 11 : General description of *GlobalContextConfigure*

Description	Invoking
By invoking this function, the user lets the method know which processor is used. The processor, as well as a function used to allocate memory, is stored by this function in global variables within a global structure. Thanks to the name of the processor used, the method will know: • The kind of processor used • The kind of handling required • The internal structure of the processor • The structure of the internal memory of the processor (if any) • The address of this internal memory • The registers of the processor (it will be possible now to read them to acquire any setting) • etc,...	Invoked only once at the very beginning of the initialization of the system.

Table 12 : Parameters of *GlobalContextConfigure*

Parameter	Explanation
processor_name	Identifier of the used processor. It is mandatory the user provides a name taken from a provided list (enum) of supported processors.
processor_internal_registers_address	This is the address of the beginning of the internal memory of the processor. Many modern communication processor features such an internal memory that enable the configuration of the processor through memory mapped registers.

UserProvidedInitializationAlloc	This is the address (equivalent to the name) of a function that enables the allocation of memory. This function can be used by the method each time any memory is needed. For example, at each channel creation, some memory will be needed for the channel_management_structure.

Table 13 : Actions to be performed by *GlobalContextConfigure*

Actions	Remark
Saves in a global structure (for example global_management_structure) the passed parameters.	

Kernel: ChannelCreate

Prototype: channel*Create*(channel_phy_type, channel_number, channel_id)

Table 14 : General description of Channel*Create (Channel Create)*

Description	Invoking
This function creates the channel. It creates the channel_management_structure associated to the channel.	This function is invoked only once, at the creation of the channel

Table 15 : Parameters of Channel*Create*

Parameter	Explanation
channel_phy_type	Specifies the PHY type of the channel. It is assumed there is a long list (enum for example) of all the supported PHY types. (for example, for one of the following Motorola's Communication processors: 360, 860, 8260; this parameter may be: SMC, SPI, SCC, MCC, FCC, etc...)

channel_number	The number given to the PHY according to the processor's reference Manual. Together channel_phy_type and channel_number completely specifies which channel the user intends to start. (For example, for a 8260, if channel_phy_type = FCC and channel_number = 2 it means the user intends to work with the FCC2 of the processor)
channel_id	This is an output of the function. This is typically the pointer to the channel_management_structure of this new channel. As already mentioned, this memory storage can be got by invoking the function *UserProvidedInitializationAlloc* (provided by the user during the system initialization when he invoked *GlobalContextConfigure* and since then stored in the global_management_structure).

Table 16: Actions to be performed by Channel*Create*

Actions	Remark
provides channel_id	Got by invoking the function *UserProvidedInitializationAlloc* (provided by the user during the system initialization when he invoked *GlobalContextConfigure* and since then stored in the global_management_structure)
fills some fields of the channel_management_structure	The PHY type of this channel (channel_phy_type) The number of this channel (channel_number)
invokes Channel*Scan* (the scanning function)	Channel*Scan* fills some other fields of the channel_management_structure

Scanner: Overview

The scanner comprises all the parts that must be aware of the processor being used. For each supported processor, the scanner knows where to find (within the registers of the processor) the information needed to initialize some fields of the channel management structure, from within the memory (when the processor registers themselves point to an internal/external memory).

The scanner, when implemented in a program, can be built as a large merge of many functions, each dependent on the processor. When the processor is known (passed as a parameter by the user), the generic function variables used by the scanner are initialized within the specific function corresponding to the passed processor. If a given processor has not been previously taken into account within the scanner, it obviously cannot be supported.

The functionality that enables reading, within a processor, of the settings (or static initializations) of a given channel is called "scanning". This scanning function assumes that the channel (link) has been previously set (the static initializations have already been performed). These static initializations can be done by any third party code. Sometimes the chipmakers themselves provide examples of such static initializations, to be used by the chip customers (typically, such useable examples are provided on the chipmaker's Web sites). The aim of these static initializations provided by the chipmakers is not to represent a full driver but rather static initializations that enable to send/receive a very limited number of data units. These initializations, if implemented alone, usually crash because they include no dynamic handling of the data units. Even if partial, these static initializations are enough, from the point of view of the chipmaker, to establish correct behavior of the chip.

Scanner: ChannelScan

Prototype: Channel*Scan*(channel_id)

Table 17 : General description of Channel*Scan*

Description	Invoking
This function fills the fields of the channel_management_structure associated to the channel (pointed to by channel_id). This function DEPENDS on the processor on which the channel runs and on the protocol it supports. We claim not particular generality in this function that must know with which processor it deals. This function can be built as a merge of a lot of smaller functions, each dependent on the processor.	This function is invoked within Channel*Create*

Table 18 : Parameters of Channel*Scan*

Parameter	Explanation
channel_id	This is an input. channel_id has been evaluated by the function Channel*Create*. This is typically the pointer to the channel_management_structure of this new channel.

Table 19 : Actions to be performed by Channel*Scan*

Actions	Remark
Determines the processor used:	Reads it from global_management_structure
Determines the address of the internal memory of the processor if any (the kind of processor is known)	Reads it from global_management_structure
Determines: • channel_phy_type • channel_number	Thanks to channel_id that points to channel_management_structure which comprises these two fields.
interim conclusion	Because of the previous steps: • The processor is known • its internal memory address (when relevant) is known • the kind of channel is known • the number of this channel is known
Invokes (thanks to a simple case, for example) a subfunction that is dedicated to the specific processor used.	Because of the previous steps, this specific subfunction knows very well in which context it works. So it can easily determine within the processor the values needed to perform the next step.

initializes the fields of the channel_management_structure.	Some fields have already been initialized by the function Channel*Create*. This subfunction initializes most if not all of the remaining fields of channel_management_structure.

Example of a typical sequence of functions calls

/* one time configuration */

GlobalContextConfigure is invoked

/* creation of a channel */

/* once per channel */

Each time a channel is to be created:

{

Using a third party code (see p35), static initializations of this channel are performed.

Channel*Create* is invoked

This function invokes Channel*Scan*

}

/* dynamic life (receive an transmit) of the channel */

/* in a neverending basis */

Each time a transmit is needed, the user invokes *TransmitFunction*

When the transmit is completed, at interrupt level, *TransmitHandler* is invoked.

Each time a data unit is received, at interrupt level, *ReceiveHandler* is invoked.

Advantages

It is easy to check from the previous descriptions of the main functions of the kernel, that nothing in the actions performed by these functions depends directly on the particular protocol/processor. All the dependencies have simply been moved into the variables and stored in the channel_management_structure of the channel. In contrast to the present method which provides a universal driver, current drivers available for the foundation layer are all written in a way the code itself is dependent on the protocol and/or the processor.

Using the present invention enables the production of a unified code because the particularities of a processor/protocol are set in the variables of the particular channels (these

variables are set during the initialization and grouped into a structure attached to each particular channel: the channel_management_structure). In other words, the code is unified in a universal driver, and the differences are grouped into the data ("data" means variables of the channel_management_structure, so it is clear the "data" can easily be initialized and modified in run time, providing all the needed flexibility to support different protocols).

There are significant advantages of such a unified code, including:

i- The same unified code can support multiple different protocols simultaneously.

ii- It is debugged once, and then services all the protocols.

iii- Each time a bug is discovered and corrected, all the protocols benefit from the debugging (the reliability continuously increases)

iv- The unified code can be installed in the instruction cache of the processor avoiding the situation of "cache miss" that occurs when several different (one per protocol) drivers compete for the instruction cache. The installation in cache of a unified code enables a situation of never-ending "cache hit", therefore boosting the real -time performance of the Layer 1-1$^{1/2}$ software.

v- Because it is unified, the software built according to the invention can be granted with advanced functional features (with non unified software, it is simply much more costly to try and do so).

It should be noted that the claim that the same code can support several different processors, refers to such functionality by the same source code. Because it is clear that different families of processors may require completely different kinds of object or executable code. The fact that a recompilation per architecture (family of processors) is required is not a major problem, since it is sufficiently advantageous that only a single unified source has to be maintained.

Graphic illustration with a Communication Processor

Figure 10 illustrates, by way of example, the way the present invention works in the case when it is implemented with a Communication Processor, in order to support three channels (links) running three different protocols: HDLC, Ethernet and ATM.

As can be seen in Figure 10:

i. In a first phase of the initialization of each channel, the scanner **124** reads **110** from the internal register **126** of the processor **125** the settings and addresses it needs.

ii. In a second phase of the initialization of each channel, the scanner **124** copies **111** this information into the different fields within the channel management structures **122**.

iii. The messages to be transmitted **101** are passed to the kernel **120** by the upper layers **121**. In fact the addresses in memory **123** of these messages are passed to the Kernel **120** (together with the channel number).

iv. The kernel **120** accesses **102** the channel management structure **122** to know where these addresses have to be copied and how to set the semaphores for transmission.

v. The kernel **120** accesses **104** the channel management structure **122** to know where to find new received messages and to know how to use the semaphore to indicate that the BDS's are free for new receptions.

vi. Furthermore, the messages that have been received are passed **103** to the upper layers **121** (in fact their addresses as well as the corresponding channel numbers).

vii. When supporting a channel, the kernel **120** accesses **105** the channel management structure **122** to get information at the required times. Using the information attained from the channel management structures **122**, the kernel **120** knows where (generally within the memory **123**) to read (receive) **108** a message from or to write (transmit) a message **108** to for this channel.

The case of a pure CPU

In the case where a pure CPU is a processor (like a simple Pentium processor) that includes no on-chip com part, the com part (that handles communication operations for the processor) is assumed to be performed by an external component. This external component may be a software component, hardware component and/or software-hardware hybrid component, as is achieved today, for example, in a standard PC motherboard. The rest of the procedure is unchanged. In this way, the com functionality of the communications processor is performed by a pure CPU.

The foregoing description of the embodiments of the invention has been presented for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise form disclosed. It should be appreciated that many modifications and variations are possible in light of the above teaching. It is intended that the scope of the invention be limited not by this detailed description, but rather by the claims appended hereto.

WHAT IS CLAIMED IS:

1. A protocol and processor independent apparatus for handling the reception and transmission of data units, comprising:

a unified transmission module, said transmission module enabling transmitting a plurality of protocol specific data units from a plurality of processor types;

a unified receive module, said receive module enabling receiving a plurality of protocol specific data units the data units from a plurality of processor types;

a unified interrupt handling module, for handling a plurality of interrupts generated by a plurality of processors, when receiving and transmitting the data units; and

a plurality of channel specific Channel Management Structures, said structures enabling managing parameters, variables and variable functions required to define behavior of particular protocol-processor channels.

2. The apparatus of claim 1, further comprising a universal data receive and data transmit manager, for enabling dual functions from said unified transmission and said unified receiving modules.

3. The apparatus of claim 1, further comprising a Configuration Layer of Static Part, for initializing channel settings in the processor.

4. The apparatus of claim 1, further comprising a scanner component for reading communication register settings and initial protocol-to-channel information from a processor's memory, thereby creating a channel management structure.

5. The apparatus of claim 1, further comprising a memory management system, for integrating at least one external Buffer Management System (BMS) into the apparatus.

6. The apparatus of claim 1, further comprising a universal tracing component for providing real time tracing capability within the apparatus.

7. The apparatus of claim 1, further comprising a universal First In First Out (FIFO) management component for enabling unified FIFO queue management for providing services selected from the group consisting of managing multiple FIFO settings for the apparatus; and providing unified FIFO management for multiple-FIFO configurations.

8. The apparatus of claim 1, further comprising a communications processor, said processor comprising:
a com part, for handling communication operations for the processor; and
a core part, for executing code in the processor.

9. The apparatus of claim 8, wherein said communications processor further comprises a hardware interrupt controller, for handling different relative priorities of the hardware events.

10. The apparatus of claim 1, further comprising a pure processor, said processor comprising:
a core part, for executing code; and
an external component for handling communication operations for said pure processor, said external component being selected from the group consisting of software components, hardware components and software-hardware hybrid components.

11. A universal driver apparatus for enabling data transmissions between a plurality of protocol-processor channels:
a universal driver for managing the dynamic handling of data units, according to common upper layer protocol functionalities of separate software drivers;
a plurality of Channel Management Structures, for managing variable data for enabling said protocol functionalities; and
lower layer code for managing the static initializations of the plurality of protocol- processor channels.

12. The apparatus of claim 11, wherein said Channel Management Structures' architecture is protocol independent and processor independent.

13. A method for creating a universal driver, the method comprising:
i. distinguishing between dynamic and static layers for a plurality of specific protocol-processor pair drivers;
ii. further distinguishing between dynamic upper layer protocol functionalities and means to execute said functionalities, in said dynamic layer of said pair drivers;
iii. transferring of said means to execute said functionalities from each said driver to a Channel Management Structure;
iv. unifying said dynamic upper layer protocol functionalities from said plurality of pair drivers into a unified driver; and
v. maintaining static layers from said plurality of drivers in a lower layer code of said universal driver.

14. The method of claim 13, wherein said unified driver is characterized by handling code that enables placing of said universal driver in an instruction cache of a processor.

15. The method of claim 13, wherein said unified driver is smaller than a typical size upper layer of a single driver.

16. A method for creating a universal Application Program Interface (API), comprising:
i. executing an initialization of a channel, by running an initialization code at the time of creation of said channel; and
ii. executing a data management code for said channel on an on-going basis, each time a data unit is handled.

17. The method of claim 16, wherein said data management code uses functional prototypes that support the main functionalities of any ongoing data management.

18. The method of claim 17, wherein said functional prototypes are characterized by the features selected from the group consisting of: protocol independent; processor independent; physical channel (PHY) independent and independent of any number used in processor documentation to identify said channel.

19. A method for enabling data transmission between a plurality of protocols and processors, the method comprising:

i. setting up a global management structure, for storing specific protocol-processor pair variables upon initializing of a specific protocol-processor channel, by a software function;

ii. configuring a static part of at least one said channel, by a configuration layer of static part module, and writing said static part to memory;

iii. scanning said memory to read at least a part of said static part of said channel, by a scanner component, and using said part to create a data management structure for supporting said channel, by said scanner component;

iv. filling said data management structure with variable parameters for said channel, by said scanner component;

v. transferring a data command to said global management structure's kernel, from said data management structure; and

vi. executing data handling for said specific protocol-processor pair, by said global management structure kernel.

FIGURES

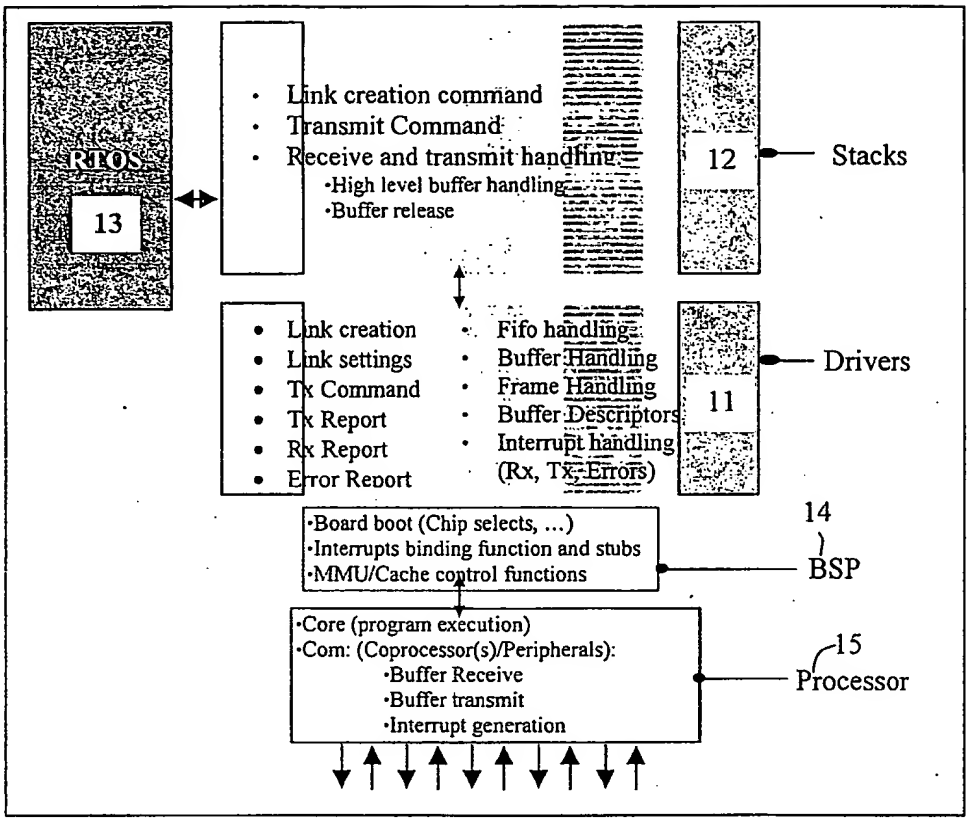


Figure 1

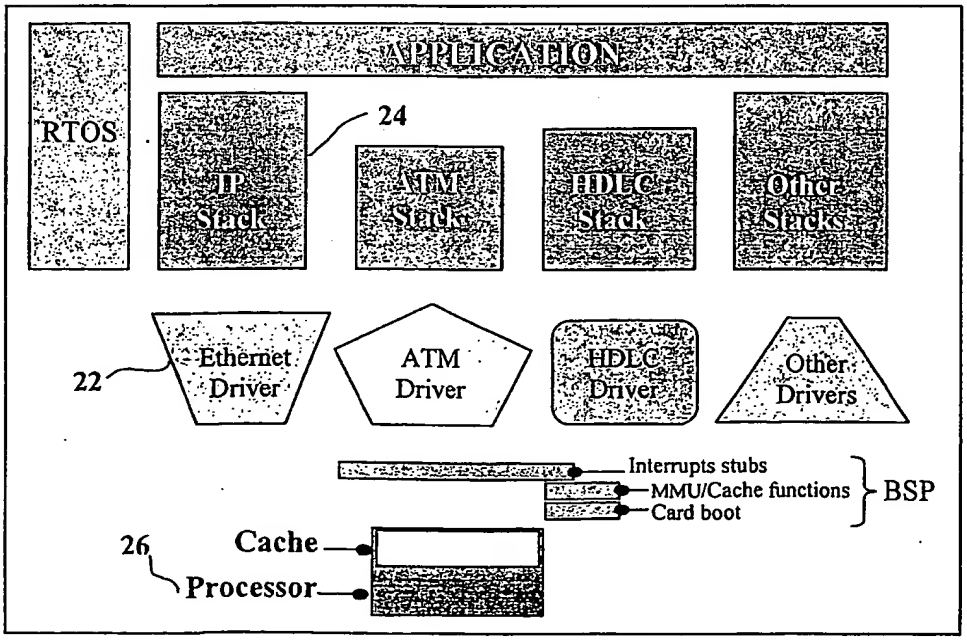
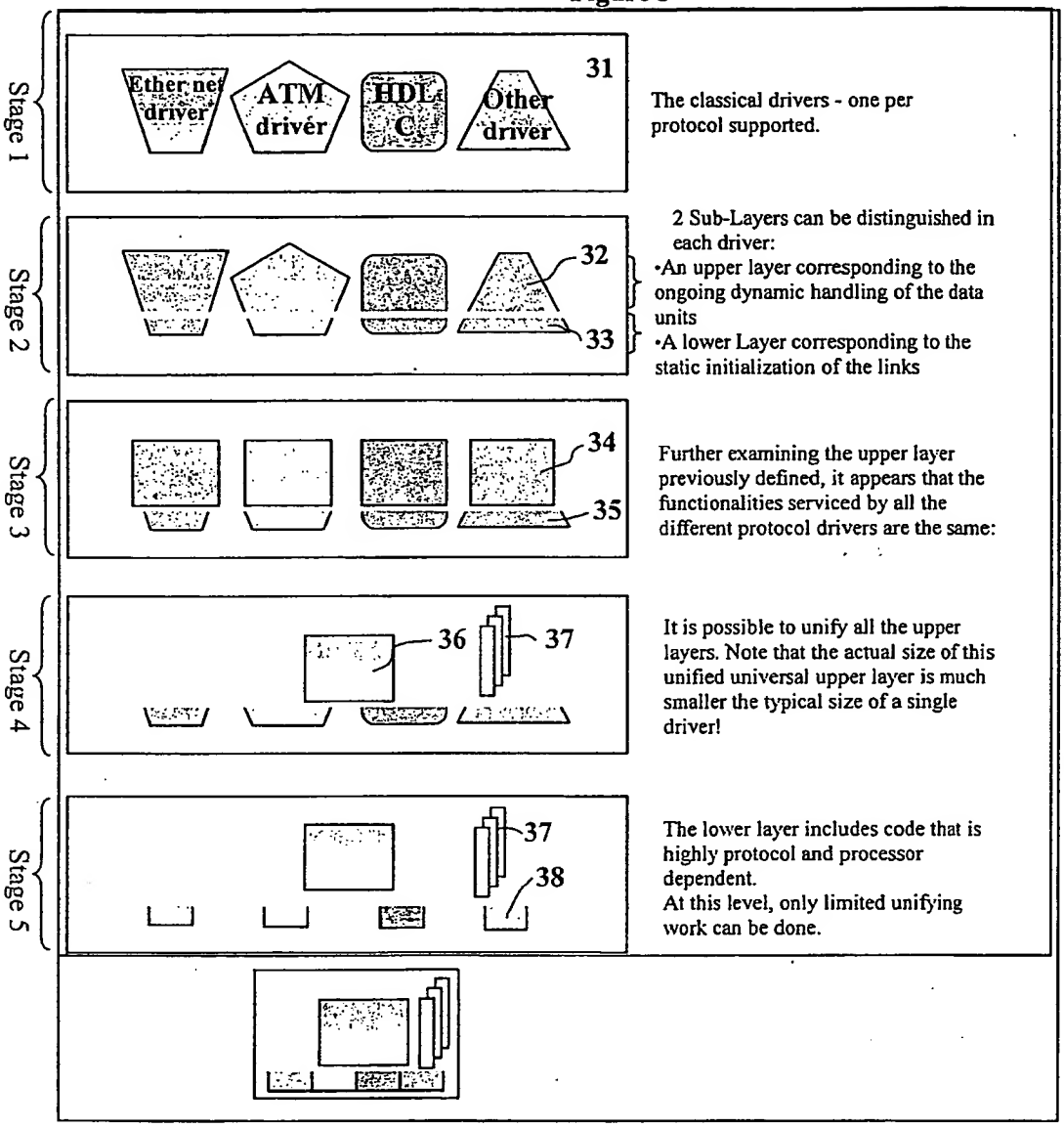


Figure 2

Figure 3


Stage 1
Ether net driver
ATM drivér
HDL C
Other driver
31
The classical drivers - one per protocol supported.
Stage 2
32
33
2 Sub-Layers can be distinguished in each driver:
•An upper layer corresponding to the ongoing dynamic handling of the data units
•A lower Layer corresponding to the static initialization of the links
Stage 3
34
35
Further examining the upper layer previously defined, it appears that the functionalities serviced by all the different protocol drivers are the same:
Stage 4
36
37
It is possible to unify all the upper layers. Note that the actual size of this unified universal upper layer is much smaller the typical size of a single driver!
Stage 5
37
38
The lower layer includes code that is highly protocol and processor dependent.
At this level, only limited unifying work can be done.

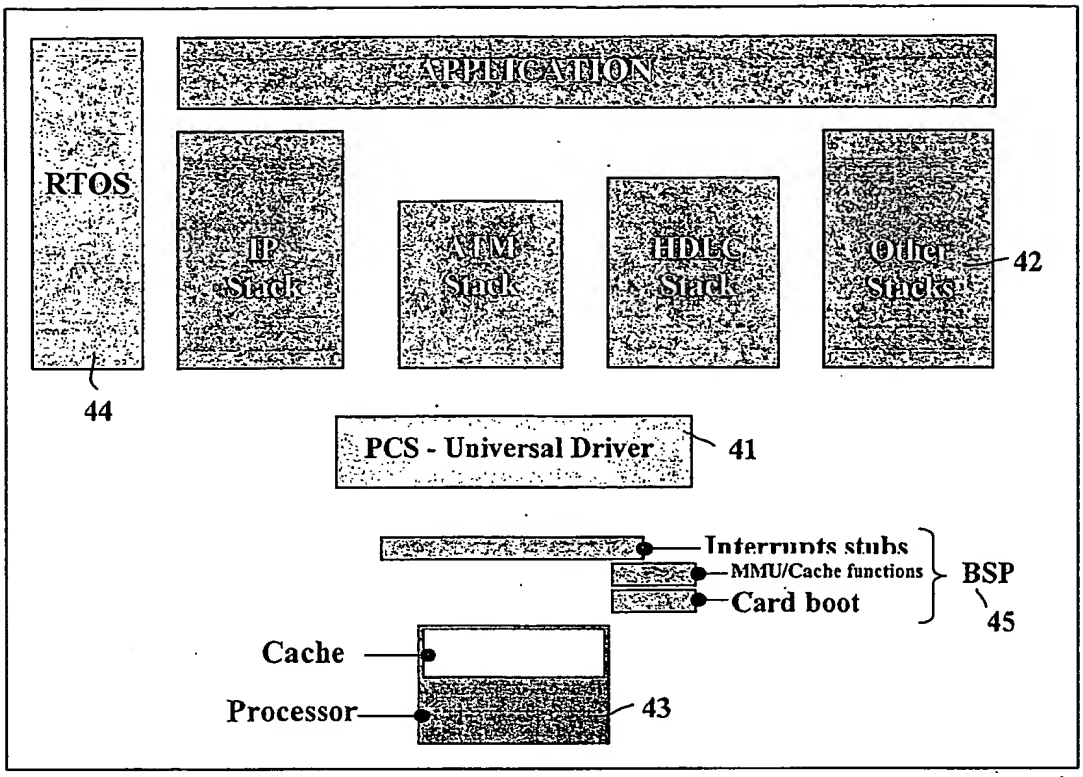

APPLICATION
RTOS
IP Stack
ATM Stack
HDLC Stack
Other Stacks
42
44
PCS - Universal Driver
41
Interrupts stubs
MMU/Cache functions
Card boot
BSP
45
Cache
Processor
43

Figure 4

Figure 5

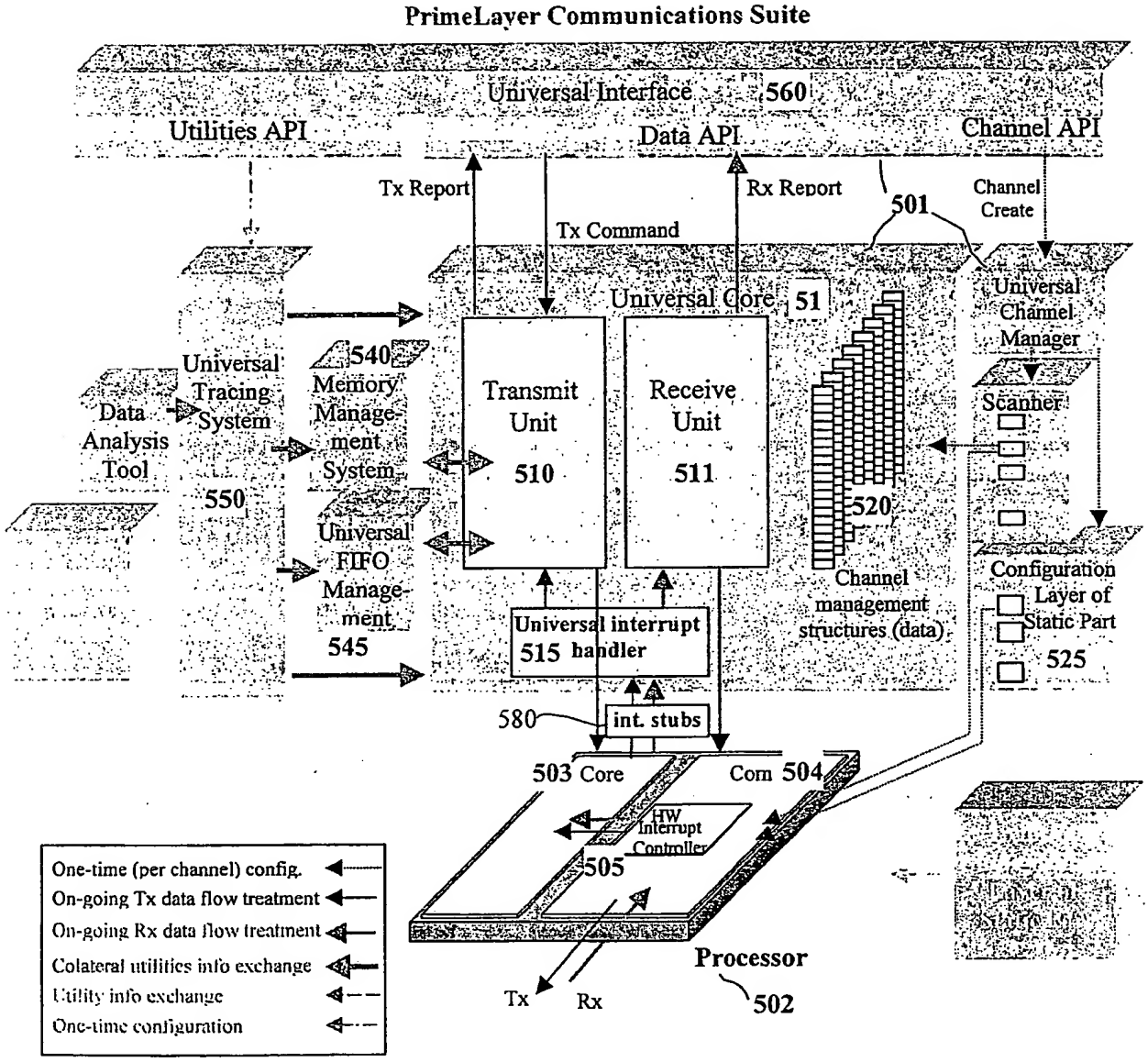

PrimeLayer Communications Suite
Universal Interface 560
Utilities API
Data API
Channel API
Tx Report
Rx Report
Channel Create
501
Tx Command
Universal Core 51
Universal Channel Manager
Universal Tracing System
Data Analysis Tool
550
540 Memory Manage-ment System
Transmit Unit 510
Receive Unit 511
520
Scanner
Universal FIFO Manage-ment 545
Channel management structures (data)
Configuration Layer of Static Part 525
Universal interrupt handler 515
580
int. stubs
503 Core
Com 504
HW Interrupt Controller
505
Processor
502
Tx
Rx
One-time (per channel) config.
On-going Tx data flow treatment
On-going Rx data flow treatment
Colateral utilities info exchange
Utility info exchange
One-time configuration

Figure 6

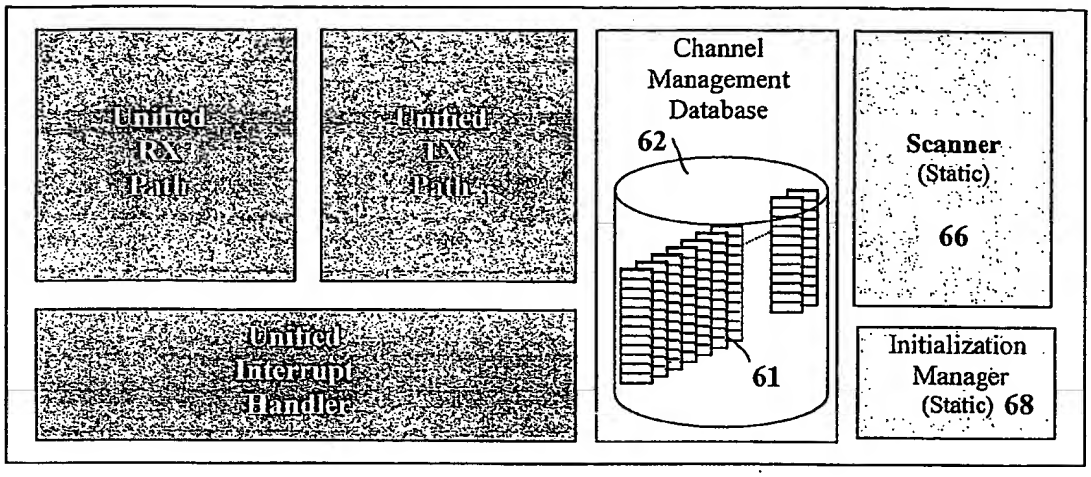


Figure 7

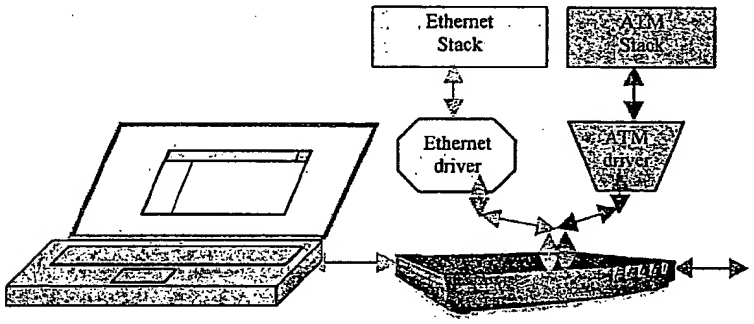


Figure 8

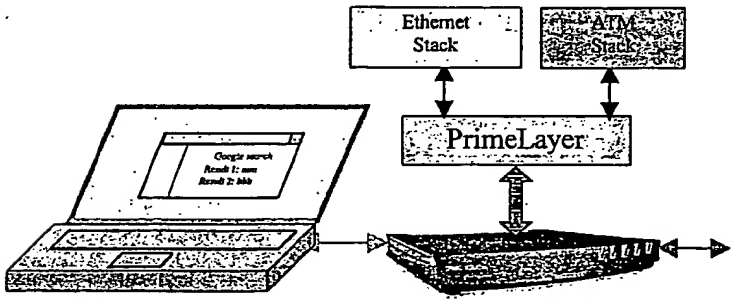
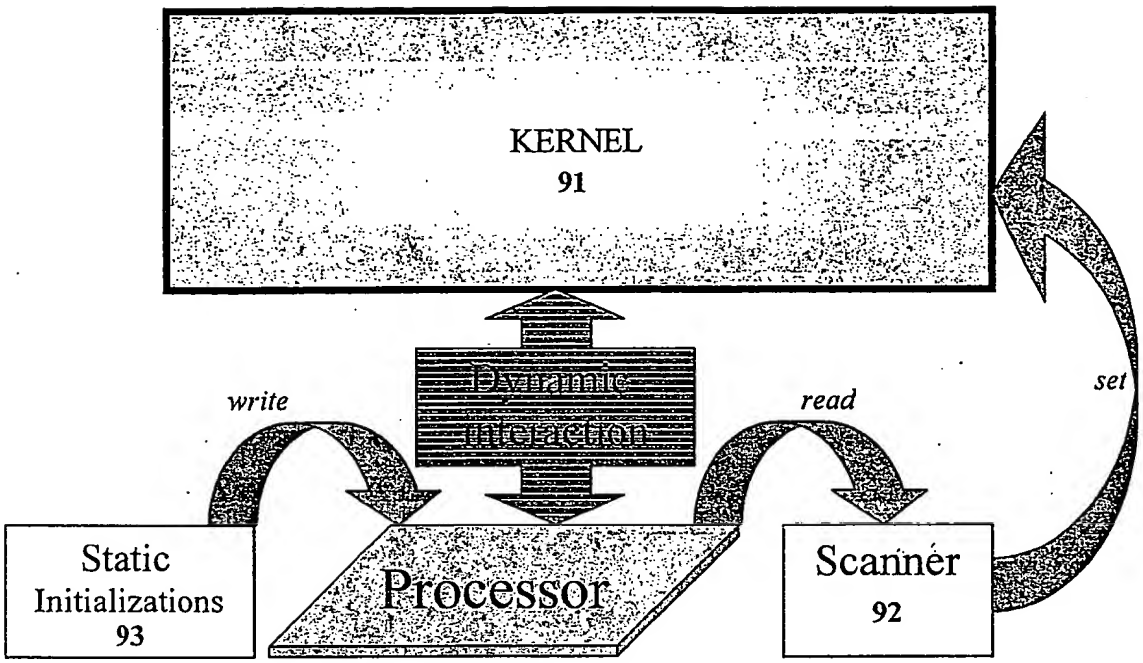


Figure 9


KERNEL
91
Dynamic
interaction
write
read
set
Static
Initializations
93
Processor
Scanner
92

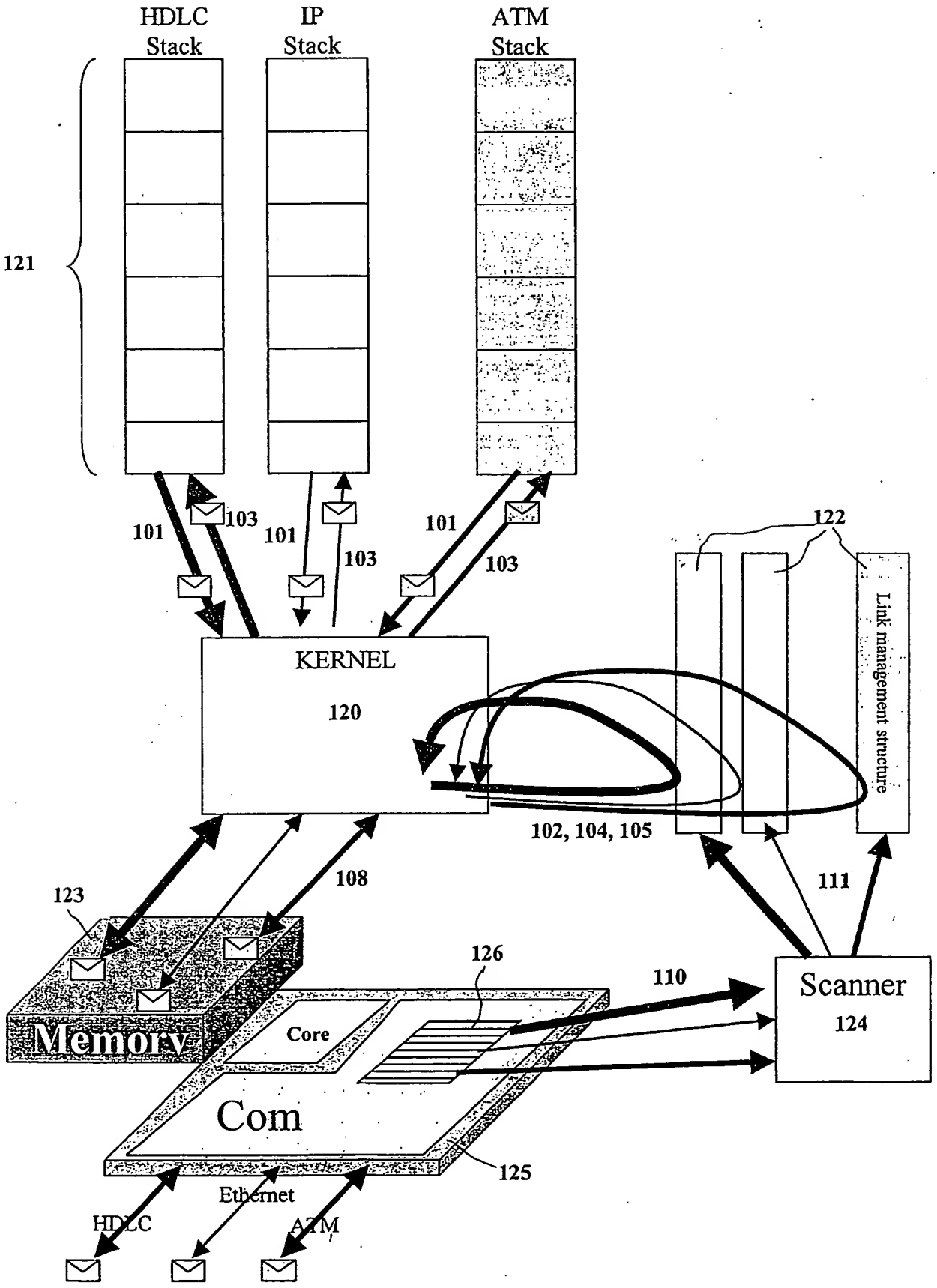

HDLC Stack
IP Stack
ATM Stack
121
101
103
101
103
101
103
KERNEL
120
122
Link management structure
102, 104, 105
108
111
123
126
110
Scanner
124
Memory
Core
Com
125
Ethernet
HDLC
ATM

Figure 10

INTERNATIONAL SEARCH REPORT

International application No.
PCT/IL02/00652

A. CLASSIFICATION OF SUBJECT MATTER

IPC(7) : G06F 9/00, 9/44, 15/00
US CL : 709/321, 302; 703/27

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
U.S. : 709/321, 206, 315

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
NONE

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EAST

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 6,148,346 A (HANSON) 14 November 2000 (14.11.2000), column 2, lines 8-63.	1-19
Y	US 6,080,202 A (STRICKLAND et al) 27 June 2000 (27.06.2000), column 1, lines 9-24.	1-19
Y	US 6,041,365 A (KLEINERMAN) 21 March 2000 (21.03.2000), column 10, lines 44-67; column 11, lines 1-47.	1-19

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
13 December 2002 (13.12.2002)	09 JAN 2003
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703)305-3230	Authorized officer Meng-Ai An Telephone No. (703)305-9669

Form PCT/ISA/210 (second sheet) (July 1998)

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☑ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☑ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: _____

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.